Методичка 4.2 - Идентификация циклов

Обзор циклов

Циклы - это единственная конструкция (за исключением оператора GOTO), которая имеет ссылку назад к более младшим адресам. Все условные конструкции, которые были разобраны ранее, были направлены только вперед к более старшим адресам. Эту особенность стоит запомнить, так по этому признаку можно определить наличие цикла в дизассемблированном листинге.

Циклы можно разделить на 3 группы по расположению условия:

- В начале (for, while)
- В конце (do-while)
- В середине (break, continue)

Условия тоже могут быть разными. Условия завершения или условие продолжения цикла. Операторы циклов for, while, do-while работают исключительно с условиями продолжения.

Циклы for и while относятся к группе циклов с предусловием и со счетчиком. На самом деле, конструкция for(i=0; i < 6; i++) { … } представляет собой тоже самое, что и конструкция i = 0; while (i < 6) { …; i++; }. Отличить данные конструкции в дизассемблированном листинге не просто, но это не всегда требуется, так как основная цель - это понять, какие операции совершаются в цикле и с какими данными.

Обзор циклов

Для демонстрации того, как выполняется идентификация циклов нам понадобится программа на языке Си, в которой будут использоваться различные виды циклов. Специально для этой цели была подготовлена программа prog-3.2.

Исходный код программы prog-3.2

```
#include <windows.h>

#pragma comment(lib,"user32.lib")

int main(int argc, char* argv[]) {

char text[] = "qwerty";
char title[] = "Prog-3.2";

for(int i = 0; i < 6; i++) {
text[i] -= 0x20;
```

```
}

MessageBox(NULL, text, title, 0);

while (text[0] != 'A') {
text[0]--;
}

MessageBox(NULL, text, title, 0);

do {
text[5]++;
} while (text[5] != 'Z');

MessageBox(NULL, text, title, 0);

return 0;
}
```

В рамках курса мы выполняем компиляцию программы prog-3.2 без оптимизации с флагом -Od для упрощения понимания конструкций, но стоит знать, что при включенной оптимизации особо умные компиляторы могут очень сильно изменить структуру цикла относительного того, что было в исходном коде на языке высокого уровня.

Например, счетчик в дизассемблированном листинге может не увеличиваться от 0 до 6, как было в исходном коде, а уменьшаться с 6 до 0. Условие при таком счетчике будет работать быстрее, так как оператор DEC, который, как вы помните, уменьшает значение на единицу, в процессе работы цикла не только установит значение 0 для переменной i, но и выставит Zero Flag в 1. Таким образом, инструкцию CMP, которая изменяет флаги состояния на основе результата сравнения, можно уже не использовать. И как результат, программа будет работать быстрее.

Также при оптимизации компилятор может изменить расположения условия цикла, например расположить его в конце цикла для цикла for. Это компилятор может сделать только в том случае, если он сможет определить по исходному коду, что цикл выполнится гарантированно хотя бы один раз.

Таким неоднозначности могут усложнить идентификацию циклов for, while и do-while в дизассмеблированном листинге.

Компиляция:

> cl prog-3.2.c -Od /Gs- /GS- /link /DYNAMICBASE:NO /NXCOMPAT:NO

Давайте посмотрим на наши циклы for, while и do-while в программе prog-3.2 под отладчиком OllyDbg.

Анализ циклов for, while и do-while

Переходим в начало секции кода (Ctrl + G, 0x00401000), где расположена единственная в программе функция main.

Ставим точку останова в начале функции (F2), чтобы при перезапуске программы сразу перейти в начало функции main.

В самом начале мы видим пролог функции, а также копирование локальных переменных из секции данных в стек через регистры EAX, ECX и DL.

Итак, мы подошли к первому циклу в программе prog-3.2.

Мы видим безусловный переход JMP с адресом, который указывает на старшие байты. После перехода мы попадаем на условие

CMP [LOCAL.1], 6

Происходит сравнение какой-то локальной переменной с числом 6. Обратите внимание, что встроенный анализатор отладчика OllyDbg догадался, что мы сравниваем число 6 именно с локальной переменной.

Если мы выключим встроенный анализ (Analysis - Remove Analysis), то увидим, что значение расположено в самом начале стекового фрейма (EBP - 4), это действительно локальная переменная.

Давайте вернем анализ обратно (Analysis - Analysis code).

Посмотрим чему равно значение локальной переменной в окне Dump (Ctrl + G, LOCAL.1). Пока значение равно нулю.

Вспоминаем, что оператор CMP работает также как SUB, т.е. происходит операция 0 - 6. В результате меняется знак и выставляется флаг SF.

Флаг SF != OF и ZF не выставлен, значит условный переход JGE не будет выполнен.

Видим, что переход JGE это условие выхода из цикла. Стрелка указывает на старшие байты за границу тела цикла.

Далее с помощью инструкции MOV мы кладем значение счетчика в регистр ECX. Это значение будет использоваться как смещение по строке. Затем записываем первый байт строки “qwerty” в

регистр EDX. Код ASCII буквы q это 0x71. Вычитаем 0x20 и получаем значение 0x51 в регистре EDX.

Теперь мы записываем новый байт в строку. Для этого получаем смещение и кладем его в регистр EAX. На данном этапе оно будет указывать на первый байт в строке. А затем с помощью инструкции MOV мы записываем байт в строку. Затем мы видим безусловный переход в начало тела цикла.
Мы попадаем на код, который выполняет инкремент счетчика и начинается новая фаза цикла.

Цикл будет работать до тех, пока счетчик не будет больше или равен 6.

Сейчас мы разобрали общую конструкцию цикла for в дизассемблированном листинге.

Затем мы попадаем на вызов функции MessageBoxA, которая покажет нам уже модифицированную строку, которая состоит только из заглавных букв.

Далее мы переходим к реализации цикла while. По общей структуре он совсем не отличается цикла for, который мы разобрали ранее.

В самом начале мы видим проверку на условие.

Задача первых трех инструкций получить указатель на байт, с которым будет осуществляться сравнение.

Здесь используется новая для нас инструкция, которую мы ранее не встречали это инструкция IMUL. В данном случае она работает с тремя операндами и кладет результат произведения второго и третьего операнда в регистр, который указан в первом операнде.

IMUL dest, src1, src2

В нашем случае, результат будет всегда 0, так как третий операнд равен нулю. Это и есть смещение до первого байта в строке. Оно у нас является постоянным.

Затем происходит сравнение байт из строки с байтом 0x41, это байт “A” большое.

Если байты совпадут, то произойдет выход из цикла, значит эта инструкция является местом выхода из цикла.

Если выхода из цикла не произошло, то следующие инструкции уменьшают первый байт строки на единицу и цикл повторяется.

Затем мы попадаем на вызов функции MessageBoxA, которая покажет нам уже модифицированную строку, первый байт которой будет равен “A” большое.

Далее мы видим цикл do-while. Его ключевое отличие в том, что условие находится в конце, т.е. цикл выполнится обязательно хотя бы 1 раз.

В самом начале мы получаем указатель на пятый байт в строке, затем записываем его в регистр AL, далее прибавляем единицу и записываем его в строку.
Далее идет подготовка к сравнению. Мы снова получаем последний байт строки в регистр EDX и выполняем сравнение с байтом 0x5A. Согласно таблице ASCII это буква ‘Z’ большое.

Сравнение проходит успешно и мы сразу выходим из цикла.

Посредством вызова функции MessageBoxA мы видим модифицированную строку, которая имеет последний байт “Z” большое. Далее переходим к эпилогу функции и функция завершает свою работу.

Циклы с условием в середине

while(TRUE)
{
...
if (condition_1) break;
…
if (condition_2) continue;
...
}

Рассмотренные нами циклы не позволяют расположить условие в середине цикла, но многие программисты идут на хитрость и используют оператор break в середине цикла, чтобы выйти из него.

Оператор break также является единственным способом выхода из бесконечного цикла, а бесконечный цикл реализуется за счет оператора JMP с адресом, который направлен к более младшим адресам. Получается, если вы при анализе встретили бесконечный цикл в программе, то стоить внимательно к нему отнестись и проанализировать весь цикл и попробовать найти условие, которое будет указывать на инструкцию, которая будет идти сразу после инструкции JMP. Если такого условия не будет, значит цикл действительно бесконечный.

Помимо условия break, в программах можно встретить оператор continue. Этот оператор перехода к условию цикла. Если цикл с предусловием, то переход будет к младшим адресам, если цикл с постусловием, то переходим к старшим адресам.

Вложенные циклы

while(TRUE)
{

```
...
while ( k < 5)
{
...
}
...
}
```

Циклы могут быть вложенными, это когда частью тела первого цикла является второй цикл. Вложенные циклы достаточно часто используются при разработке программ, следовательно, их тоже необходимо уметь анализировать.

Начало цикла можно определить по перекрестной ссылке, которая направлена вниз. Конец цикла - это условный или безусловных переход на его начало. Важно запомнить, что у цикла может быть только одно начало и один конец. Причем, что важно, циклы не могут пересекаться. Следовательно, если между началом и концом одного цикла, встречается начало другого цикла, то этот цикл будет вложенным.

Трансляция цикла с условием в середине

При анализе вложенным циклов есть нюансы. Например, при использовании оператора continue в циклах с предусловием.

В этом примере оператор continue транслируется в безусловный переход и направлен вверх, поэтому его сложно отличить от “конца” цикла.

Два “конца” и два “начала” вполне напоминают два цикла, где один вложен в другой. Но при этом начала обоих циклов совмещены. Такое может быть только в том случае, если в цикл с постусловием вложен цикл с предусловием.

Вот в этом и есть основная сложность идентификации вложенных циклов.

В этом примере все очень похоже на наличие вложенного цикла, но мы видели исходный код и знаем, что здесь вложенного цикла быть не должно.

Давайте попробуем разобраться.

Если посмотреть внимательно, то можно увидеть, что при невыполнении первого условия переход будет выполнен на “конец” цикла, т.е. за вторую границу. Если бы это было предусловием вложенного цикла, то прыжок был бы на первый “конец”, а не на второй.

Получается, что это не два цикла, а один. А первый “конец” это результат выполнения операции continue.